# Extending a HSF-enabled Open-Source RTOS with Resource Sharing

– International Workshop OSPERT 2010 –

**M.M.H.P. van den Heuvel** - R.J. Bril - J.J. Lukkien - M. Behnam

System Architecture and Networking (SAN)
Department of Mathematics and Computer Science
Eindhoven University of Technology
The Netherlands

6 July 2010

Technische Universiteit **Eindhoven** University of Technology

SAN

MÄLARDALEN UNIVERSITY SWEDEN

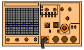
MRTC

# Outline


1. Introduction
2. Task Synchronization
3. Inter-subsystem resource sharing
4. Conclusions

# Embedded Real-time Systems

Many embedded devices provide **multiple, integrated functionalities**.



In such systems its important to deliver correct functionality **on time**.

**Non-real-time systems**

- Correct function if produced result is correct

System does the right thing

**Real-time systems**

- Correct function if produced result is correct and **delivered on time**

System does the right thing...

+

...and it does it on time

These functionalities share both **logical** and physical **resources**.

# Integration Problem

1. Isolation: applications shall not *interfere*
   - Temporal isolation (processor);
   - Spatial isolation (memory).

# Integration Problem

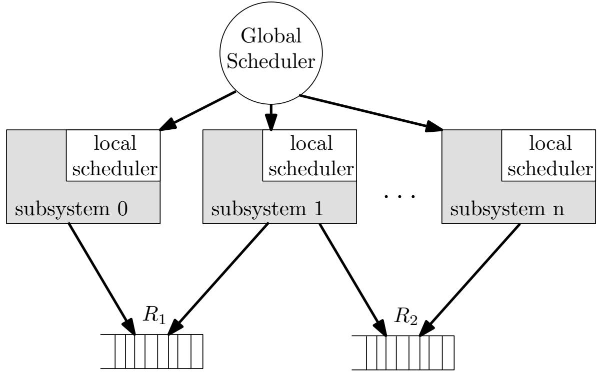
1. Isolation: applications shall not *interfere*
   - Temporal isolation (processor);
   - Spatial isolation (memory).

2. Development and analysis versus integration
   - Independent analysis of application on *virtual platforms*;
   - Application specific scheduling algorithms;
   - Composition of virtual platforms.

# Integration Problem

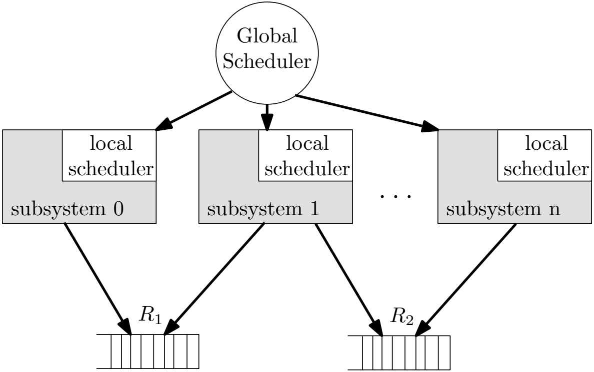
1. Isolation: applications shall not *interfere*
   - Temporal isolation (processor);
   - Spatial isolation (memory).
2. Development and analysis versus integration
   - Independent analysis of application on *virtual platforms*;
   - Application specific scheduling algorithms;
   - Composition of virtual platforms.
3. Applications may share logical resources.

# A Solution: Hierarchical Scheduling




- subsystem: *server*, *set of tasks* and *local (task) scheduler*
- server: a budget allocated each *period*

# A Solution: Hierarchical Scheduling




- subsystem: *server*, *set of tasks* and *local (task) scheduler*
- server: a budget allocated each *period*

**Tasks, located in arbitrary subsystems, may share logical resources**

# Outline

1 Introduction

2 Task Synchronization

3 Inter-subsystem resource sharing

4 Conclusions

# MicroC/OS-II Basics

MicroC/OS-II is

- a commercial RTOS
- targeted at embedded systems
- open source
- available at `http://micrium.com/`

It provides

- a portable and configurable kernel
- a fixed-priority, preemptive task scheduler
- basic services (mailboxes, **mutexes** and counting semaphores)

# GRASP: microC/OS-II Instrumentation and Visualization

- Visualization of scheduling behavior:

  M. Holenderski, M. van den Heuvel, R. J. Bril, and J. J. Lukkien.
  Grasp: Tracing, visualizing and measuring the behavior of real-time systems.
  In *1st WATERS*, July 2010.

- MicroC/OS-II port to OpenRISC platform

- OpenRISC: Architectural Simulator
  `http://opencores.org/openrisc,or1ksim`

# microC/OS-II's mutexes: Priority calling

Priority calling is similar to *priority inheritance protocol* (PIP):

**Priority Inheritance Rule:**

when a higher-priority task blocks on a resource,
the lower-priority task holding the resource inherits the higher priority;

# microC/OS-II's mutexes: Priority calling

Priority calling is similar to *priority inheritance protocol* (PIP):

**Priority Inheritance Rule:**

when a higher-priority task blocks on a resource,
the lower-priority task holding the resource inherits the higher priority;

| Priority Calling | Priority Inheritance |
|---|---|
| unique priority for each resource | run-time priority adaption |
| inherits a predefined priority | transitive priority inheritance |
| non-transparent | transparent |

# microC/OS-II's mutexes: Priority calling

Priority calling is similar to *priority inheritance protocol* (PIP):

**Priority Inheritance Rule:**

when a higher-priority task blocks on a resource,
the lower-priority task holding the resource inherits the higher priority;

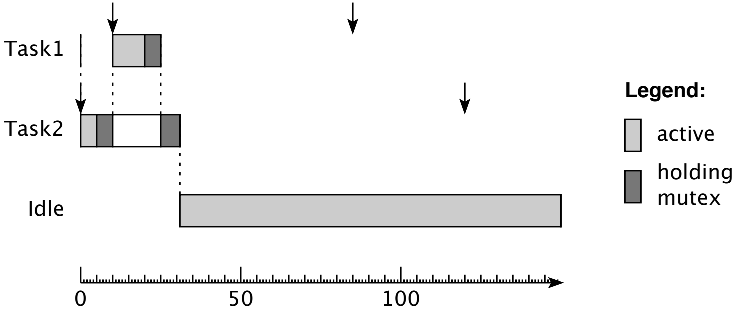
| Priority Calling | Priority Inheritance |
|---|---|
| unique priority for each resource | run-time priority adaption |
| inherits a predefined priority | transitive priority inheritance |
| non-transparent | transparent |

**It is not**: Highest Locker Protocol (HLP) or Stack Resource Policy (SRP).

- microC/OS-II: a task inherits a higher priority *only* when a higher priority task is blocked;
- in HLP/SRP a task immediately inherits a priority *when it locks a resource*.

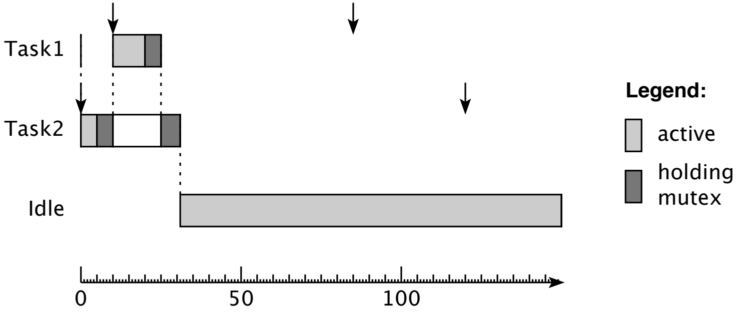
# microC/OS-II: protocol classification

microC/OS-II's synchronization protocol suffers from deadlock:



# microC/OS-II: protocol classification

microC/OS-II's synchronization protocol suffers from deadlock:




**Conclusion:**

microC/OS-II implements a *non-transparent, priority-inheritance protocol*

# Intermezzo: Stack Resource Policy (SRP)

- Each resource has a statically determined *resource ceiling*:

**Definition of a resource ceiling:**

the maximum priority of any task that could use the resource.

# Intermezzo: Stack Resource Policy (SRP)

- Each resource has a statically determined *resource ceiling*:

**Definition of a resource ceiling:**

the maximum priority of any task that could use the resource.

- A dynamically updated *system ceiling* is maintained:

**Definition of the system ceiling**

the maximum resource ceiling of any resource currently being locked in the system.

# Intermezzo: Stack Resource Policy (SRP)

- Each resource has a statically determined *resource ceiling*:

**Definition of a resource ceiling:**

the maximum priority of any task that could use the resource.

- A dynamically updated *system ceiling* is maintained:

**Definition of the system ceiling**

the maximum resource ceiling of any resource currently being locked in the system.

- A task can only be selected for execution if
  1. it has the highest priority among all ready tasks;
  2. its priority is higher than the current system ceiling.

# Intermezzo: SRP (Continued)

- SRP provides non-blocking primitives:
  - therefore it allows tasks to share their execution stack;
  - blocking occurs upon an attempt to preempt, rather than upon an attempt to access a resource.

# Intermezzo: SRP (Continued)

- SRP provides non-blocking primitives:
  - therefore it allows tasks to share their execution stack;
  - blocking occurs upon an attempt to preempt, rather than upon an attempt to access a resource.

- SRP is non-transparent,
  similar as microC/OS-II's PIP-like implementation.

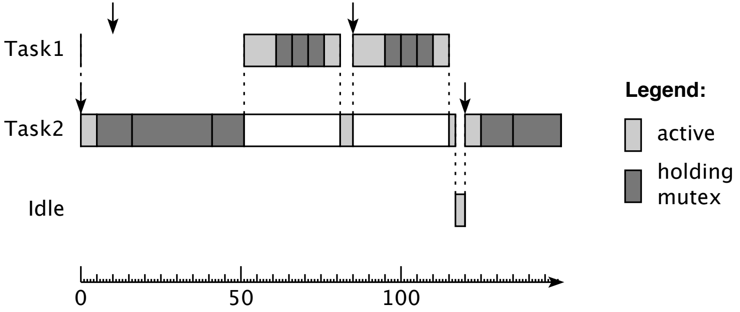
# Intermezzo: SRP (Continued)

- SRP provides non-blocking primitives:
  - therefore it allows tasks to share their execution stack;
  - blocking occurs upon an attempt to preempt, rather than upon an attempt to access a resource.

- SRP is non-transparent, similar as microC/OS-II's PIP-like implementation.

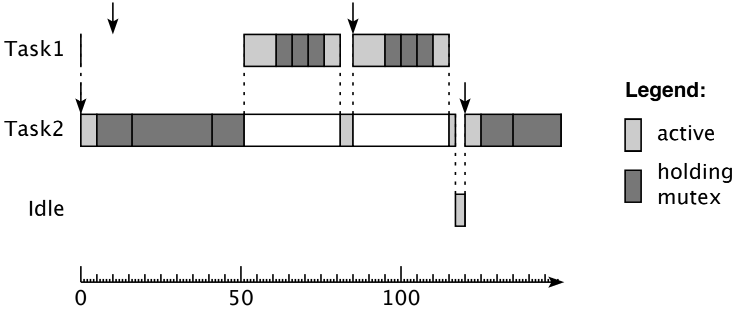
- Maintaining the system ceiling can be implemented using a stack data structure:
  - we stack the *resource ceilings* of used resources in a *monotonically increasing* order;
  - the top of the stack represents the *system ceiling*.

Deadlocks are resolved:



# microC/OS-II: Our SRP extension

Deadlocks are resolved:




- Easy implementations (approx. 170 lines of code);
- Extended microC/OS-II scheduler with SRP's preemption rule.

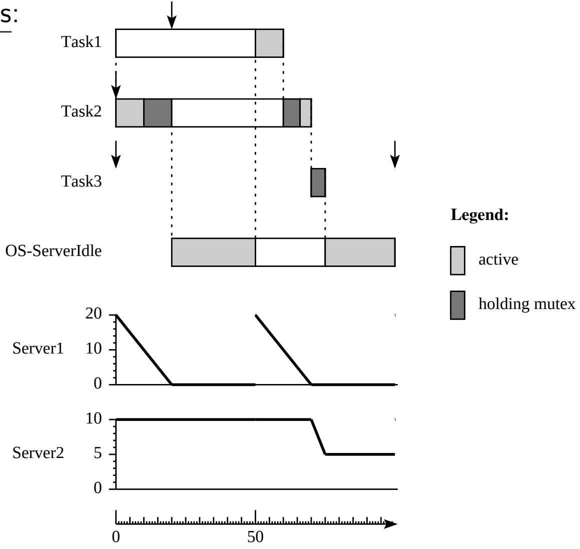
# Outline


1. Introduction
2. Task Synchronization
3. Inter-subsystem resource sharing
4. Conclusions

# Global resource sharing problem

*Budget depletion* during a critical section can lead to excessive blocking times:

- SRP locally
- SRP globally



# Global resource sharing problem

Two SRP-based solutions for fixed-priority scheduling:

- **HSRP:** React upon budget depletion while a resource is locked; i.e. allow to use an overrun budget
  1. **with payback:** the consumed overrun budget is subtracted from the next budget provisioning;
  2. **no payback:** no penalty for overrun consumption.

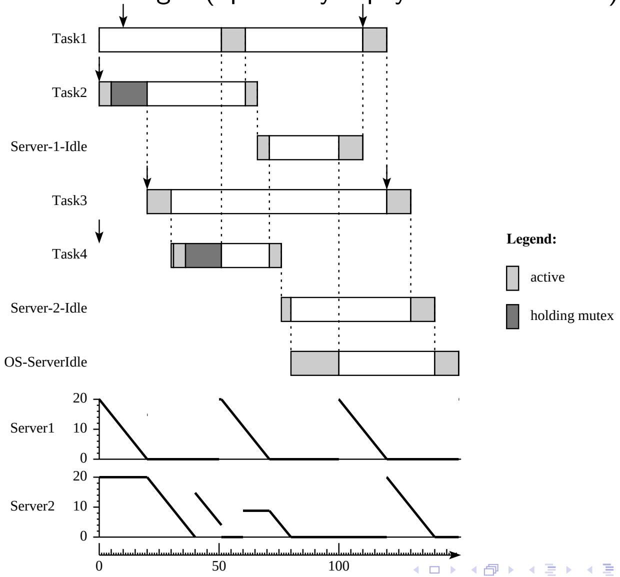
# Global resource sharing problem

Two SRP-based solutions for fixed-priority scheduling:

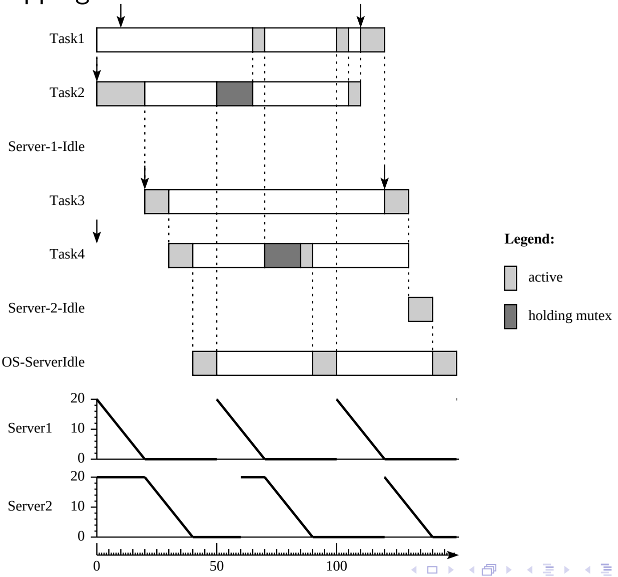
- **HSRP:** React upon budget depletion while a resource is locked; i.e. allow to use an overrun budget
  1. **with payback:** the consumed overrun budget is subtracted from the next budget provisioning;
  2. **no payback:** no penalty for overrun consumption.
- **SIRAP:** Prevent budget depletion during resource access; i.e. before granting access, first check the remaining budget.

# microC/OS-II: HSRP extension

HSRP provides overrun budget (optionally a payback mechanism):



SIRAP uses a skipping mechanism:



# HSRP and SIRAP implementation overhead and issues

| **Event** | **HSRP** | **SIRAP** |
|---|---|---|
| Lock resource | - | spinlock |
| Unlock resource | overrun completion | - |
| Budget depletion | overrun | - |
| Budget replenishment | overrun completion, payback (optionally) | spinlock-completion |

# HSRP and SIRAP implementation overhead and issues

| Event | HSRP | SIRAP |
|---|---|---|
| Lock resource | - | spinlock |
| Unlock resource | overrun completion | - |
| Budget depletion | overrun | - |
| Budget replenishment | overrun completion, payback (optionally) | spinlock-completion |

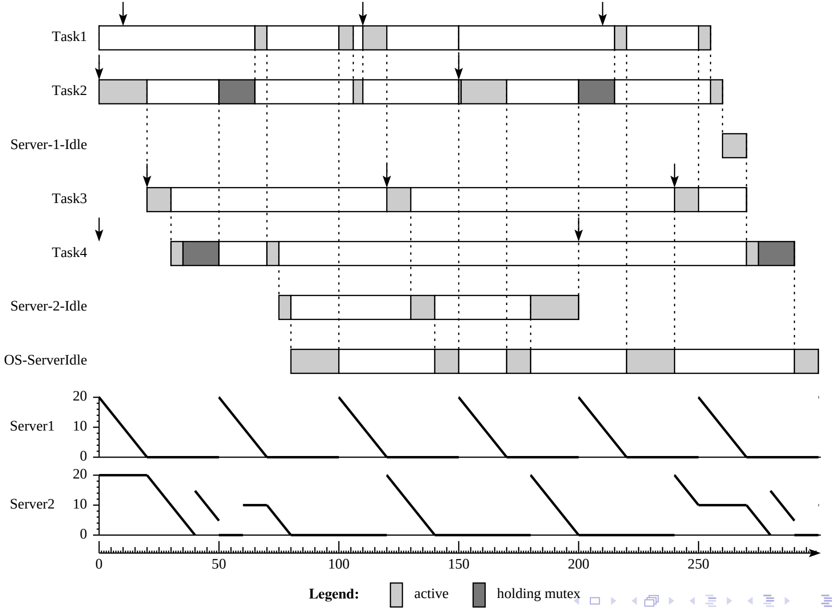
- **HSRP**:
  - close to default SRP;
  - expensive queue manipulations to track overrun budget;
  - complex implementation due to explicit event handling.

# HSRP and SIRAP implementation overhead and issues

| **Event** | **HSRP** | **SIRAP** |
|---|---|---|
| Lock resource | - | spinlock |
| Unlock resource | overrun completion | - |
| Budget depletion | overrun | - |
| Budget replenishment | overrun completion, payback (optionally) | spinlock-completion |

- **HSRP**:
  - close to default SRP;
  - expensive queue manipulations to track overrun budget;
  - complex implementation due to explicit event handling.
- **SIRAP**:
  - spinlocking is executed within a task's context, but wastes budget;
  - alternatively: suspend (i.e. block) and resume a task, but this is not SRP-compliant!

# HSRP and SIRAP side-by-side: Unified interfaces



# Outline

1 Introduction

2 Task Synchronization

3 Inter-subsystem resource sharing

4 Conclusions

# Conclusions

We presented:

- a classification of microC/OS-II's synchronization protocol;
- an efficient task-level SRP implementation;
- two alternative hierarchical SRP-implementations, i.e. SIRAP and HSRP;
- a side-by-side integration of SIRAP and HSRP in a single HSF.

# Conclusions

We presented:

- a classification of microC/OS-II's synchronization protocol;
- an efficient task-level SRP implementation;
- two alternative hierarchical SRP-implementations, i.e. SIRAP and HSRP;
- a side-by-side integration of SIRAP and HSRP in a single HSF.

We made a minimal number of modifications to MicroC/OS-II.

# Conclusions

We presented:

- a classification of microC/OS-II's synchronization protocol;
- an efficient task-level SRP implementation;
- two alternative hierarchical SRP-implementations, i.e. SIRAP and HSRP;
- a side-by-side integration of SIRAP and HSRP in a single HSF.

We made a minimal number of modifications to MicroC/OS-II.

Upcoming work:

- EDF-based synchronization (including BROE);
- protocol-transparent global resource sharing.

# References


M. Åsberg, M. Behnam, T. Nolte, and R. J. Bril.
Implementation of overrun and skipping in VxWorks.
In *6th OSPERT*, July 2010.

M. Behnam, I. Shin, T. Nolte, and M. Nolin.
SIRAP: A synchronization protocol for hierarchical resource sharing in real-time open systems.
In *EMSOFT*, October 2007.

R. I. Davis and A. Burns.
Resource sharing in hierarchical fixed priority pre-emptive systems.
In *27th RTSS*, December 2006.

M. M. H. P. van den Heuvel, R. J. Bril, and J. J. Lukkien.
Protocol-transparent resource sharing in hierarchically scheduled real-time systems.
In *15th IEEE ETFA*, September 2010.